

## Overview

- If buffer overflow vulnerabilities could be completely eliminated a large proportion of the most serious security threats would disappear with them
- Here we briefly survey some of the various mitigating strategies proposed to deal with the problem of buffer overflow vulnerabilities
- Material is largely drawn from:
  - *Buffer Overflows: Attacks and Defences for the Vulnerability of the Decade*, by Cowan et al
  - *Secure Coding in C and C++*, by Seacord
  - *Secure Programming with Static Analysis*, by Chess & West



## Buffer Overflow Goals

- The goal of a buffer overflow attack is to subvert program flow and if that program has sufficient privileges, to control the host machine
- Buffer overflow attacks dominate the class of remote network penetration attacks
- An attacker usually has two subgoals:
  1. Place suitable code in the process address space
  2. Jump to that code
- To achieve the first goal an attacker can inject or use code already present in process memory



## Buffer Overflow Goals

- To achieve the second goal an attacker overflows a buffer to corrupt adjacent program state
- In overflowing a local buffer the attacker's usual target is the active frame's return address: this is a traditional stack smashing attack
- A simple return address may not always be the target, function pointers may also be targeted and rewritten to point elsewhere



## Buffer Overflow Goals

- Also, `longjmp` buffers may be corrupted to cause execution to jump to an arbitrary location where the usual `setjmp(buffer)` and `longjmp(buffer)` sequence is exploited (used in stack unwinding)
- If the overflowing buffer contains sufficient space the injected code is normally placed there but, in general, any location will do as long as the attacker can jump to it e.g. in some other buffer, in an environment variable etc.



## Buffer Overflow Defences

1. Write correct code (make use of static analysis)
2. Operating system approach (make the stack non-executable)
3. Direct compiler approach (incorporate bounds checking)
4. Indirect compiler approach (use StackGuard)
5. Other approaches (e.g. build behaviour models or use instruction encryption)



## Writing Correct Code

- A worthy but expensive proposition especially in error-prone C/C++
- Despite a long-established understanding of C/C++ coding vulnerabilities, new security threats continue to emerge
- Simple tools such as `grep`, code auditing teams, fault injection utilities all help
- More sophisticated analysis tools such as “Fortify” and “Coverity” employ static analysis and progress continues to be made
- However, C’s semantics and the subtle nature of many overflows mean none of the above approaches can guarantee security



## NX

- Here the idea is a simple one: make segments other than the text section non-executable
- A little trickier than it would first appear:
  - Operating systems will dynamically insert code into data segments to support optimisation
  - Code to handle signals will be placed on the stack
- Nonetheless, hardware NX support can be exploited to effectively disable execute permission everywhere but in the text section
- This approach offers no protection against attacks that use existing code or exploit a vulnerability to modify data



## Bounds Checking

- This approach completely removes the threat of buffer overflow as program state can never be corrupted
- All reads and writes require monitoring to ensure they are within bounds
- Has been implemented in Compaq’s C compiler and invoked with the `check_bounds` option
- Unfortunately its limitations are quite severe. . .



## Bounds Checking

- Only explicit array references (i.e. `[]`) are checked
- Subroutine array references are not checked (i.e. cannot see an array beyond where it is declared)
- Invoked `libc` functions are not checked
- Jones and Kelly wrote a patch for `gcc` which does the same thing, it performs full array reference checking
- Performance costs are substantial: some matrix multiplication operations were slowed by a factor of 30
- Programs can be linked against Purify libraries to ensure only legitimate array references but, again, at a performance cost: a slow down factor of between 3 and 5 is incurred depending on the context



## Code Pointer Integrity Checking

- Instead of preventing overflow as with bounds checking we accept overflows may occur and instead of trying to prevent them we attempt to detect the attack afterwards but before overwritten code pointers are dereferenced
- One disadvantage is that overflows that affect program state other than code pointers will go undetected
- The main advantage over bounds checking is in the area of performance, the hit is small
- Snarskii has developed a custom Free BSD `libc` in which hand-coded assembly checks were added to ensure activation record (stack frames to us) integrity before dereferencing



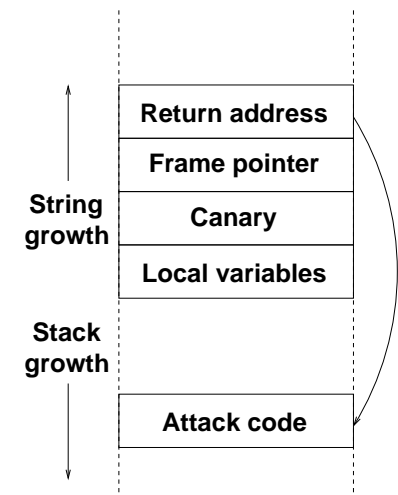
## Code Pointer Integrity Checking

- Fine as far as `libc` goes but will not protect against vulnerabilities in new code
- StackGuard comes as a `gcc` patch that instruments frame prologue and epilogues
- The enhanced set up code places a “canary” value near the return address and the enhanced teardown code checks its integrity before returning
- Traditional stack smashing is thus thwarted
- Canary forgery is prevented using one of two alternatives:
  1. Using a terminator canary that contains NULL, CR etc.
  2. Using a random 32-bit canary that is hard to guess



## Code Pointer Integrity Checking

- StackGuard has been proven to provide an effective defence against existing and new buffer overflow attacks
- System compatibility and performance are preserved e.g. it has been successfully combined with `ssh` and `Apache`



## Generalised Pointer Integrity Checking

- PointGuard is a generalised version of StackGuard which allows for the placement of canaries next to arbitrary security sensitive pointers or variables
- Advantages: low performance hit/implementation effort and compatible with existing code
- Disadvantage: programmer must flag relevant data
- The combination of StackGuard and a non-executable stack will protect against all forms of attack that involve frame corruption and all forms of attack in which the attack code resides on the stack thus greatly reducing the attack surface



## Anomaly Detection

- A program is executed and its behaviour recorded: behaviour may be expressed in terms of some observable events such as system calls
- An finite state automaton (FSA) is constructed based on logged behaviour
- Once deployed “in the wild”, attempted deviations from the FSA are deemed attacks and the process is shutdown gracefully
- However, as usual false positives and negatives are a problem
- “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviours” by Sekar et al provides more information



## Randomising Instruction Sets

- Executables are encrypted with a secret key and program instructions decrypted on-the-fly during program execution
- Decrypting injected code will (usually) not produce valid program instructions
- Obvious drawback is the performance hit and only defends against those attacks where code is injected into the process address space
- “Randomized Instruction Set Emulation” by Barrantes et al provides more information

